Linux 设备驱动编写教程(待整理)

作者:杨鹏飞

目录

目录..1
USB 驱动编写..2
块设备驱动编写..10
I2C 驱动编写..14
Spi 驱动编写..25
DMA 驱动程序的编写(参考韦东山的驱动代码)..33

USB 驱动编写

1.usb 驱动程序的编写：分析 linux 内核的 usb 骨架代码 usb_skeleton.c

说明:一个 usb 设备有若干个配置组成，每一个配置又可以有多个接口，每一个接口又有多个设置

而接口本身又有可能没有端点或者多个端点。usb 的数据交换是通过端点进行的。

usb 的接口分为控制接口、中断接口、批量接口、等时接口。

在 linux 中端点使用数据接口 struct usb_host_endpoint 来描述 usb 的端点信息

在 linux 中接口使用 struct usb_interface 来描述。

在 linux 中整个 usb 设备可以使用 struct usb_device 结构来描述。

在 linux 的 usb 驱动程序中，ehci 每次发送的数据包大小最好为 512，根据 linux 内核代码说明得出的

```
/* MAX_TRANSFER is chosen so that the VM is not stressed by allocations >
PAGE_SIZE and the number of packets in a page is an integer 512 is the largest
possible packet on EHCI */
```

usb 驱动程序编写流程:

1.注册 usb 子系统 usb_register(&skel_driver);

说明:skel_driver 是描述驱动信息的结构体

定义一个 id_table 结构体，来描述内核允许该模块所支持的硬件设备

//不带背景色的代码

```
#define USB_SKEL_VENDOR_ID      0xfff0
#define USB_SKEL_PRODUCT_ID     0xfff0
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
	{ USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
	{ }					/* Terminating entry */
};
```

这个结构体的最后一个元素必须为空，用来标识结束符

MODULE_DEVICE_TABLE(usb, skel_table);

module device table 用来描述设备类型，如果是 usb 设备，自然是 usb,如果是 pci 设备那么就是 MODULE_DEVICE_TABLE(pci, skel_table);

//带背景色的代码

```
static struct usb_driver skel_driver = {
	.name =		"skeleton",	/*  驱动名称  */
```

```
	.probe = skel_probe,		/*  操作函数   */
	.disconnect =	skel_disconnect, /*  断开函数  */
	.suspend =	skel_suspend,
	.resume =	skel_resume,
	.pre_reset =	skel_pre_reset,
	.post_reset = skel_post_reset,
	.id_table =	skel_table,
	.supports_autosuspend = 1,
};
```

其中 name probe disconnect 以及 id_table 是必须的,其他的可以看情况添不添加。id_table 用来告诉内核该模块支持的设备，usb 子系统通过设备的 protect id 和 vendor id_table
的组合或者设备的 class、subclass 和 protocol 的组合来识别设备。

如果一个 usb 设备接入到系统中，他的 protect id 和 vendor id 符合我们注册的，那么久会调用这个模块
作为该设备的驱动程序。

2.定义一个描述该驱动以及拥有所有资源和状态的结构体，这个结构体是程序员自己按照驱动需求定义的

```
struct usb_skel {
	struct usb_device*udev;			/* the usb device for this device */
	struct usb_interface	*interface;		/* the interface for this device */
	struct semaphore limit_sem;		/* limiting the number of writes in progress */
	struct usb_anchor	submitted;		/* in case we need to retract our submissions */
	unsigned char			*bulk_in_buffer;	/* the buffer to receive data */
	size_t			bulk_in_size;		/* the size of the receive buffer */
	__u8			bulk_in_endpointAddr;	/* the address of the bulk in endpoint */
	__u8			bulk_out_endpointAddr;	/* the address of the bulk out endpoint */
	int			errors;			/* the last request tanked */
	int			open_count;		/* count the number of openers */
	spinlock_t		err_lock;		/* lock for errors */
	struct kref		kref;
	struct mutex		io_mutex;		/* synchronize I/O with disconnect */
};
```

分析:这个结构体拥有一个描述 usb 设备的结构体 udev,一个接口 interface,一个用于接收数据的输入缓冲 bulk_in_buffer
	以及大小 bulk_in_size，批量的输入输出端口地址 bulk_in_endpointAddr、bulk_out_endpointAddr。这些变量是一个

usb 驱动程序所必须使用到的

3.编写 probel 函数 static int skel_probe(struct usb_interface *interface, const struct usb_device_id *id)

当调用这个函数的时候，系统会传递一个 usb_interface 和 id 他们分别是该 usb 设备的接口描述符(一般会是该设备的第 0 号接口，该接口的 0 号配置，以及设备描述 id--protect id 和 vendor id)

skel_probe 函数主要需要实现下面的一些代码程序:

```
dev->udev = usb_get_dev(interface_to_usbdev(interface));
```

说明:interface_to_usbdev(interface)　　/* 得到该 usb 的设备描述符 */

usb_get_dev 函数是增加对该 usb_device 的引用计数的，我们应该在操作该设备的时候增加引用计数，释放 usb 的时候减少引用计数

注意:这里的引用计数是对 usb_device 的计数，不是对模块的计数,在编写驱动程序的时候我们可以不进行计数。

```
dev->interface = interface;        /* 得到该设备的接口描述符 */
```

在得到 usb_device 之后，我们需要对 usb_skel 结构体进行初始化，这部分工作的任务主要是向 usb_skel 注册该 usb 设备的端点

```
	/* set up the endpoint information */
	/* use only the first bulk-in and bulk-out endpoints */
	iface_desc = interface->cur_altsetting; /* 获取当前接口设置 */
	for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
		endpoint = &iface_desc->endpoint[i].desc; /* 得到端点描述符 */

		if (!dev->bulk_in_endpointAddr &&
		     usb_endpoint_is_bulk_in(endpoint)) { /* 检查端点的类型和输出方向
*/
			/* we found a bulk in endpoint */
			/* 如果我们发现他是输入的端点，那么就注册到 usb_skel 中 */
			buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
			dev->bulk_in_size = buffer_size;
			dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
			dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);     /* 分
配输出缓存 */
			if (!dev->bulk_in_buffer) {
				err("Could not allocate bulk_in_buffer");
				goto error;
			}
		}

		/* 如果检查到该 usb 的端点类型是输出端点 保存输出端点的信息到
usb_skel 中*/
		if (!dev->bulk_out_endpointAddr &&
		     usb_endpoint_is_bulk_out(endpoint)) {
```

```
				/* we found a bulk out endpoint */
				dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
			}
			if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
			err("Could not find both bulk-in and bulk-out endpoints");
			goto error;
		}
		}
```

说明:在 usb_host_interface 结构体里面有一个 usb_interface_descriptor 的叫做 desc 的成员，它用来描述 interface 的一些属性，bNumEndpoints 是一个 8 位的数字，代表端口的端点数，proble 函数便利所有的端点，检查他们的类型和方向 ,并且注册到 usb_skel 中 端点是根据 HID 规范来的(什么是 HID 规范)

```
usb_set_intfdata(interface, dev); /*  向系统注册  usb_skel*/
```

说明:这个函数注册的数据结构是任意的，只是为了我们以后用 usb_get_intfdata 可以得到我们想要的数据结构

接下来我们可以注册一个文件操作函数集，这个才是 usb 设备的真正操作函数集合

```
static struct usb_class_driver skel_class = {
	.name =		"skel%d",
	.fops =		&skel_fops,
	.minor_base =	USB_SKEL_MINOR_BASE,
};

/* we can register the device now, as it is ready */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
	/* something prevented us from registering this driver */
	err("Not able to get a minor for this device.");
	usb_set_intfdata(interface, NULL);
	goto error;
}
```

4.usb 设备断开函数

当设备被拔出集线器的时候，usb 子系统会自动的调用 disconnect 函数，它主要是注销 class_device

```
static void skel_disconnect(struct usb_interface *interface)
{
	struct usb_skel *dev;
	int minor = interface->minor;
```

```
	dev = usb_get_intfdata(interface);
	usb_set_intfdata(interface, NULL);

	/* give back our minor */
	usb_deregister_dev(interface, &skel_class);

	/* prevent more I/O from starting */
	mutex_lock(&dev->io_mutex);
	dev->interface = NULL;
	mutex_unlock(&dev->io_mutex);

	usb_kill_anchored_urbs(&dev->submitted);

	/* decrement our usage count */
	kref_put(&dev->kref, skel_delete);

	dev_info(&interface->dev, "USB Skeleton #%d now disconnected", minor);
}
```

usb 操作函数集接口初始化流程说明:

```
static const struct file_operations skel_fops = {
	.owner =	THIS_MODULE,
	.read =		skel_read,
	.write =	skel_write,
	.open =		skel_open,
	.release =	skel_release,
	.flush =	skel_flush,
};
```

在这个结构体里面的操作函数集主要是对 usb 设备的具体的操作，包括数据的发送与接收,usb 数据的发送与接收是通过 urb 进行的

urb 好比高速公路上的汽车，他可以运载 usb 的 4 种数据流，不过你需要向告诉司机李需要运什么，目的地是什么。

1.创建 urb usb_alloc_urb(0, GFP_KERNEL);

第一个函数是等时包的数量，如果不是承载的等时包，应该为 0,第二个参数与 kmalloc。

要释放一个 urb 可以用:usb_free_urb 函数

要承载数据，我们还需要告诉司机目的地址信息和需要运送的货物，对于不同的数据，系统提供了不同的函数(4 种)

下面是一个 urb 使用的方法(批量传输)

```
usb_fill_bulk_urb(urb, dev->udev,
		  usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
		  buf, writesize, skel_write_bulk_callback, dev);
```

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;/*  不要再让 HCD 做 DMA 映射，相当于 DMA 映射已经做好了  */
usb_anchor_urb(urb, &dev->submitted);

/* send the data out the bulk port */
retval = usb_submit_urb(urb, GFP_KERNEL);
```

说明:skel_write_bulk_callback 是回调函数我们可以在 urb 传输完成之后通过 status 判断 urb 是否正常传输成功
如果不成功，正确的做法应该是重新提交 urb。

```
static void skel_write_bulk_callback(struct urb *urb)
{
	struct usb_skel *dev;
	dev = urb->context;
	/* sync/async unlink faults aren't errors */

	/*  正确的做法是如果 urb 传输错误应该重新提交 urb */
	if (urb->status) {
		printk("urb error!\r\n");    /*  如果 urb 提交错误  */
	}
	else
		printk("urb success!\r\n");      /*  如果正常提交 urb */

}
```

usb_sndbulkpipe:这个函数是根据设备和端点生成管道，这里的管道是批量发送管道，对应的还有

```
#define usb_sndctrlpipe(dev,endpoint)  控制发送管道
#define usb_rcvctrlpipe(dev,endpoint)  控制接收管道
#define usb_sndisocpipe(dev,endpoint)等时发送管道
#define usb_rcvisocpipe(dev,endpoint) 等时接收管道
#define usb_sndbulkpipe(dev,endpoint)      批量发送管道
#define usb_rcvbulkpipe(dev,endpoint)批量接收管道
#define usb_sndintpipe(dev,endpoint)  中断发送管道
#define usb_rcvintpipe(dev,endpoint)  中断接收管道
```

说明:一个参数是 urb,第二个参数是 usb 的设备,第三个参数是管道号码 pipe。该管道记录了目标设备的端点以及管道的类型
，每一个管道只有一种类型和一个方向，它与他的目标设备的端点相对应，
usb_sndbulkpipe 函数可以把指定的 usb 设备的指定端点
这里指定成为一个批量 out 端点(还有类似的函数可以设置成为控制 out 端点，控制 in 端点，批量 in 端点，批量 out 端点，中断 in 端点，
中断 out 端点，等时 int 端点，等时 out 端点，等时 in 端点),第四个参数是需要发送的数组，第五个参数是发送数组的长度，
第六个参数是发送完成之后的回调函数，当发送完成之后，这个函数会被自动调

用，一般在这个函数中，可以通过判断 urb->status 的值，来判断成功传输与否。第六个参数是用户自己定义的，是可能在回调函数中使用的数据。

注意:在 write 中申请的部分资源，比如说每次调用 write 都会申请的资源，例如 urb 和 buff，强烈建议在发送完成的回调函数中手动释放。

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;      /*  这句话的意思就是使用 dma 传输  */
usb_submit_urb(urb, GFP_KERNEL); /*  启动 usb 传输  */
```

事实上，如果数据量不大，那么可以不一定需要用卡车来运货，系统还提供了一种不用 urb 的传输方式，而 usb_skeleton 的读操作正式采用了这种方式实现。

```
/* do a blocking bulk read to get data from the device */
retval = usb_bulk_msg(dev->udev,
                usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                dev->bulk_in_buffer,
                min(dev->bulk_in_size, count),
                &bytes_read, 10000);

/* if the read was successful, copy the data to userspace */
if (!retval) {
    if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
        retval = -EFAULT;
    else
        retval = bytes_read;
}
```

函数原型:int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
            void *data, int len, int *actual_length, int timeout)

说明:这个函数会阻塞等待数据的传输完成或者超时，data 是输入/输出缓冲,len 是它的长度大小，actual_length 是实际传输的数据的大小，timeout 设置超时，这个函数也是批量传输数据的 usb 发送函数，只不过没有使用 urb。

以上是发送 urb，如果是需要接收数据，那么只需要调用接收 urb 函数即可

```
retval = usb_bulk_msg(dev->udev,
                usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
```

```
                    dev->bulk_in_buffer,
                    min(dev->bulk_in_size, count),
                    &bytes_read, 10000);
```

以上就是 usb 发送和接收的全部流程。

块设备驱动编写

块设备驱动程序:

块设备读写数据的基本单位是块，例如一个磁盘通常为一个 sector(扇区)

block_device 结构代表了内核中的一个块设备，他可以标识整个磁盘或者一个特定的分区。当这个结构代表一个分区的时候，它的 bd_contains 成员指向包含这个分区的设备，bd_part 成员指向设备的 gendisk 结构。

gendisk 是一个单独的磁盘驱动器的内核表示，内核还使用 gendisk 来表示分区。

gendisk 结构的操作函数集合包含了一下几个:alloc_gendisk /* 分配磁盘 */ add_disk /* 增加磁盘信息 */ unlink_gendisk /* 删除磁盘信息 */ delete_partition /* 删除分区 */ add_partition /* 添加分区 */

block_device_operations 类似于支付设备驱动里面的 file_operations 结构,它是块设备的对应接口，

这里面定义的是块设备的操作函数集

函数原型:

```
struct block_device_operations {
	int (*open) (struct block_device *, fmode_t);
	int (*release) (struct gendisk *, fmode_t);
	int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
	int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
	int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
	int (*direct_access) (struct block_device *, sector_t,
						void **, unsigned long *);
	int (*media_changed) (struct gendisk *);
	int (*revalidate_disk) (struct gendisk *);
	int (*getgeo)(struct block_device *, struct hd_geometry *);
	struct module *owner;
};
```

说明:

```
static struct block_device_operations ramblock_fops = {
	.owner	= THIS_MODULE,
	.getgeo	= ramblock_getgeo,
};
```

系统对块设备进行读取操作时，是通过块设备通用的读写操作函数将一个请求保存在该设备的操作请求队列中，
然后调用这个块设备的底层处理函数，对请求队列中的操作请求进行逐一执行，request_queue 结构描述了块设备的请求队列。
请求队列拥有一系列相关的处理函数: 创建队列时提供了一个自旋锁 blk_init_queue、获取队列中第一个未完成的请求 elv_net_request、请求完成 end_request、请求停止 blk_stop_queue、开始请求 blk_start_queue、清除请求队列 blk_clenup_queue

块设备驱动程序编程:

1.初始化请求队列

```
blk_init_queue(do_ramblock_request, &ramblock_lock);
```

说明:该函数的第一个参数是请求处理函数的指针，第二个参数是控制访问队列的权限的自旋锁(在编写对应的驱动程序的时候，一定要检测返回值)。
请求处理函数不能由驱动自己调用，只有当内核认为是时候让驱动处理对设备的读写等操作时，它才会调用这个函数。
对于 ramdisk 这种完全随机访问的非机械设备，并不需要复杂的 I/O 调度，这个时候，可以直接踢开"I/O 调度器"，使用如下的函数来绑定请求队列和制造请求的函数(make_request_fn)

```
void blk_queue_make_request(struct request_queue *q, make_request_fn *mfn)
```

说明:blk_alloc_queue 和 blk_queue_make_request 结合起来使用的逻辑一般是

```
xxx_queue = blk_alloc_queue(GPF_KERNEL)
blk_queue_make_request(xxx_queue, xxx_make_request)
```

2.注册块设备

```
register_blkdev(0, "ramblock");    /* cat /proc/devices */
```

说明:块设备的注册和支付设备一样的，有自己的设备名字和设备号，如果主设备号为 0，那么内核会自动分配一个新的主设备号，
一般情况下，默认写 0。该函数正常返回的是主设备号，就是 major 的值

3.分配一个 gendisk 结构(gendisk 在 linux 内核中用来标识一个独立的设备或者分区)

```
alloc_disk(16); /*  次设备号个数: 分区个数+1 */
```

说明:alloc_disk 的参数 代表次设备号，同一个磁盘的各个分区共享一个主设备号，而此设备号则不相同，所以这里的此设备号可以用来代表分区的个数，因为次设备号是从 0 开始的，所以是参数+1 个分区。

4.当我们分配好 gendisk 结构后，我们需要初始化 gendisk，设置其属性

1).设置 gendisk 的请求队列 ramblock_disk->queue = ramblock_queue;

2).设置主设备号 ramblock_disk->major   = major;

3).分配次设备号 ramblock_disk->first_minor = 0; /* 相当于分区的起始号 在此

后的程序中该值不能够修改*/
4).指定块设备的操作函数集合 ramblock_disk->fops   = &ramblock_fops;
5).设置 gendisk 的 name 字段 sprintf(ramblock_disk->disk_name, "ramblock");
6).设置扇区大小 set_capacity(ramblock_disk, RAMBLOCK_SIZE / 512); /* 配置各个扇区的容量 */
7).注册磁盘设备(gendisk 结构分配之后还不能马上被使用)add_disk(ramblock_disk);

说明:在块设备的卸载过程中完成与模块加载函数相反的工作
1)清除请求队列，使用 blk_cleanup_queue
2)删除对 gendisk 的引用，使用 put_disk
3)删除对块设备的引用，注销块设备驱动，使用 unregister_blkdev

5.在我们初始化 gendisk 结构体之后，我们需要填充 file_ops 结构
主要的操作函数说明:
1)获取驱动器信息 int (*getgeo)(struct block_device *, struct hd_geometry *);(疑问:这个函数在什么时候被调用？)

```
static int ramblock_getgeo(struct block_device *bdev, struct hd_geometry *geo)
{
    /*  这些值得设置是按照我们分配的内存大小设置出来的  */
    /*  容量=heads*cylinders*sectors*512 */
    geo->heads         = 2;
    geo->cylinders = 32;
    geo->sectors      = RAMBLOCK_SIZE/2/32/512;
    return 0;
}
```

6.编写 request_queue 请求函数
1)提交请求 blk_peek_request(struct request_queue *q)
说明:上述函数返回一个要处理的请求(由 I/O 调度器决定)，如果没有请求则返回NULL，它不会清楚请求，而是仍然将这个请求保留在队列上，原来的老函数 elv_next_request 函数已经不再使用了(新版本的内核才会使用提交请求 blk_peek_request，如果内核中没有这个函数，请使用老版本的函数 elv_next_request)。

2)启动请求
void blk_start_request(struct request *req)
说明:从请求队列中移除请求，原先老的 API blkdev_dequeue_request()会在 blk_start_request()内部被调用。
我们可以考虑是用 blk_fetch_request()函数，它同时做完了 blk_peek_request 和 blk_start_request 的工作。

3)报告完成

它拥有一系列的 blk_end_xxx 函数，如果我们使用了 blk_queue_make_request 绕开了 io 调度，那么在 bio 出来完成之后使用 bio_endio()函数来通知处理结束。

I2C 驱动编写

i2c 驱动编写：参考韦东山的 i2c 驱动代码和 linux 内核自带的驱动代码

i2c 驱动主要分为 linux i2c 总线驱动和设备驱动，总线驱动一般已经提供好了的，我们主要是开发对应的具体外设的设备驱动

说明:编写 i2c 设备驱动有两种方法，一种是利用系统给我们提供的 i2c-dev.c 来实现一个 i2c 适配器来控制 i2c 设备，我们需要在应用程序去封装数据，我们需要做的就是在应用层来填充 i2c_msg 结构体来向驱动层传递数据。另一种是为了 i2c 设备，独立编写一个设备驱动程序。说明：在后面一种情况下，不需要使用 i2c-dev.c。

利用 i2c-dev.c 作为适配器，进而控制 i2c 设备。i2c-dev.c 并没有针对特定的设备而实现，只是提供了通用的 read、write 和 ioctl 等接口应用层可以借用这些接口访问挂载在 i2c 适配器上面的 i2c 设备。

说明:该文件只适合单开始信号，对于多开始信号需要单独编写 i2c 设备驱动程序

下面分析 i2c 的普通驱动，不使用 i2c-dev.c

注意！！！！！！！！！！！！！！！！！！！！！！！！！！！！

第一个设备驱动的写法:在板级中填充关于 i2c 设备的硬件信息，i2c_client 是内核根据板级代码自动生成的不需要我们单独来填充 i2c_client 结构。

1.填充板级信息，使用 i2c_board_info 结构体

```
struct i2c_board_info {
	char		type[I2C_NAME_SIZE];
	unsigned short	flags;	/* 标志位 */
	unsigned short	addr;	/* 设备地址 */
	void		*platform_data; /* 用来传递私有的数据 */
	struct dev_archdata	*archdata;	/* 也可以用来传递私有的数据 */
	int		irq; /* 中断号 */
};
```

说明:i2c_board_info 必须初始化两个字段，分别是设备名和设备地址，对应了 i2c_client 的 name 以及 i2c_client 的 addr。

说明:type 是设备名称，它和驱动程序的 name 字段进行匹配，使用方式类似于下面的(从 linux 内核摘录)

```
static struct twl4030_platform_data sdp3430_twldata = {
	.irq_base	= TWL4030_IRQ_BASE,
	.irq_end	= TWL4030_IRQ_END,

	/* platform_data for children goes here */
	.bci		= &sdp3430_bci_data,
	.gpio		= &sdp3430_gpio_data,
	.madc		= &sdp3430_madc_data,
	.keypad		= &sdp3430_kp_data,
	.usb		= &sdp3430_usb_data,

	.vaux1		= &sdp3430_vaux1,
	.vaux2		= &sdp3430_vaux2,
	.vaux3		= &sdp3430_vaux3,
	.vaux4		= &sdp3430_vaux4,
	.vmmc1		= &sdp3430_vmmc1,
	.vmmc2		= &sdp3430_vmmc2,
	.vsim		= &sdp3430_vsim,
	.vdac		= &sdp3430_vdac,
	.vpll2		= &sdp3430_vpll2,
};
static struct i2c_board_info __initdata sdp3430_i2c_boardinfo[] = {
	{
		I2C_BOARD_INFO("twl4030", 0x48),	/*  提示:i2c 设备本来的地址是 8
位，但是 linux 和裸机程序是有一定的区别的，这里不能够使用 8
位来代表设备地址，一般用高 7 位来代表 i2c 的设备地址，一般真实的地址右移
一位得到 linux 的 i2c 的设备地址  */
		.flags = I2C_CLIENT_WAKE,
		.irq = INT_34XX_SYS_NIRQ,
		.platform_data = &sdp3430_twldata,
	},
};
```

2.注册平台设备

板子上没有一个 i2c 设备，那么我们就需要注册一个 i2c 平台设备，注册函数如下:

```
i2c_register_board_info(1, sdp3430_i2c_boardinfo,
ARRAY_SIZE(sdp3430_i2c_boardinfo));
```

说明:第一个参数是 bus 号，第二个参数是 board_info 结构，第三个参数是大小。

如果按照以上的这种方式填充，那么我们不需要填充 i2c_client 结构，这个结构

会被操作系统根据 i2c_board_info 结构自动填充。

注意:i2c 的设备 i2c_client 不是在 i2c_register_board_info 的时候填充的，是在创建 i2c_adapter 的时候填充的，它会根据我们写的板级信息来填充。同时也会初始化 adapter 结构，一个 i2c 的设备想要进行工作，有三个过程是必不可少的，创建 i2c_cilent，创建 i2c_adapter 创建 i2c_driver。

```
struct i2c_client {
	unsigned short flags;		/* div., see below		*/
	unsigned short addr;		/* chip address - NOTE: 7bit	*/
					/* addresses are stored in the	*/
					/* _LOWER_ 7 bits		*/
	char name[I2C_NAME_SIZE];
	struct i2c_adapter *adapter;	/* the adapter we sit on	*/
	struct i2c_driver *driver;	/* and our access routines */
	struct device dev;		/* the device structure		*/
	int irq;			/* irq issued by device		*/
	struct list_head list;		/* DEPRECATED */
	struct list_head detected;
	struct completion released;
};
```

3.注册 i2c_driver 和匹配对应的 i2c_client(一个 i2c_client 对应一个相应的 i2c 设备，i2c_client 携带的是设备相关的硬件信息)

每一个 i2c 的设备驱动，必须首先创建一个 i2c_driver 结构体，该结构体包含了 i2c 设备探测和注销的一些基本的方法和信息
我们在编写对应的设备驱动的时候主要填充以下字段

```
static const struct i2c_device_id ds1682_id[] = {
	{ "at24cxx", 0 },
	{ }
};
static struct i2c_driver at24cxx_driver = {
	.driver = {
		.name	= "at24cxx", /*  驱动名称  */
	},
	.probe = at24cxx_probel, /*  匹配函数  */
	.id_table = ds1682_id
	.command = NULL,
};
```

编写好结构体信息之后，需要在模块加载函数中注册

```
i2c_add_driver(&at24cxx_driver);
```

使用 i2c_add_driver 向内核注册 i2c_驱动。一旦驱动和设备文件相匹配，那么 probel 函数将会被调用。

说明:一般情况下只需要定义 probel 函数以及 remove 函数即可，

4.在 proble 函数中我们需要定义操作函数集(字符驱动函数集)

```
static struct file_operations at24cxx_fops = {
	.owner = THIS_MODULE,
	.read   = at24cxx_read,
	.write = at24cxx_write,
};


struct i2c_client *at24cxx_client;  /*  需要提前定义好，在驱动程序中  */
static int at24cxx_probel(struct i2c_client *client,
			const struct i2c_device_id *id)
{
	int rc;
	at24cxx_client = client;   /*  说明，这里的 client 是内核根据我们填充的板级信息自动生成的  */
	major = register_chrdev(0, "at24cxx", &at24cxx_fops);

	cls = class_create(THIS_MODULE, "at24cxx");
	class_device_create(cls, NULL, MKDEV(major, 0), NULL, "at24cxx"); /* /dev/at24cxx */
	return rc;
}
```

5.定义 i2c 的 write 和 read 函数

```
static ssize_t at24cxx_read(struct file *file, char __user *buf, size_t size, loff_t * offset)
{
	unsigned char address;
	unsigned char data;
	struct i2c_msg msg[2];
	int ret;
	if (size != 1)
		return -EINVAL;

	copy_from_user(&address, buf, 1);
```

```
	/*  数据传输三要素: 源,目的,长度  */

	/*  读 AT24CXX 时,要先把要读的存储空间的地址发给它  */
	msg[0].addr    = at24cxx_client->addr;    /*  目的  */
	msg[0].buf     = &address;                /*  源  */
	msg[0].len     = 1;                       /*  地址=1 byte */
	msg[0].flags = 0;                         /*  表示写  */

	/*  然后启动读操作  */
	msg[1].addr    = at24cxx_client->addr;    /*  源  */
	msg[1].buf     = &data;                   /*  目的  */
	msg[1].len     = 1;                       /*  数据=1 byte */
	msg[1].flags = I2C_M_RD;                          /*  表示读  */


	/*  启动传输  */
	ret = i2c_transfer(at24cxx_client->adapter, msg, 2);
	if (ret == 2)
	{
		copy_to_user(buf, &data, 1);
		return 1;
	}
	else
		return -EIO;
}


static ssize_t at24cxx_write(struct file *file, const char __user *buf, size_t size, loff_t
*offset)
{
	unsigned char val[2];
	struct i2c_msg msg[1];
	int ret;

	/* address = buf[0]
	 * data        = buf[1]
	 */
	if (size != 2)
		return -EINVAL;

	copy_from_user(val, buf, 2);

	/*  数据传输三要素: 源,目的,长度  */
	msg[0].addr    = at24cxx_client->addr;    /*  目的  */
```

```
	msg[0].buf      = val;                          /*  源  */
	msg[0].len      = 2;                            /*  地址+数据=2 byte */
	msg[0].flags = 0;                           /*  表示写  */

	ret = i2c_transfer(at24cxx_client->adapter, msg, 1);
	if (ret == 1)
		return 2;
	else
		return -EIO;
}
```

注意！！！！！！！！！！！！！！！！！！！！！！！！！！！！！

第二个设备驱动的写法:我们在 i2c 驱动程序手动填写关于 i2c 设备相关的硬件信息(设备地址，访问的时钟频率等)

1.定义一个 struct i2c_client *at24cxx_client;结构体，用来描述硬件信息。(在程序中进行填充)

定义一个 i2c_driver 结构体，因为这里不存在平台设备和驱动匹配的情况，所以不需要 probel 函数。

```
static struct i2c_driver at24cxx_driver = {
	.driver = {
		.name	= "at24cxx",
	},
	.attach_adapter = at24cxx_attach,  /*    */
	.detach_client	= at24cxx_detach,/*    *
}
```

定义一个 i2c_client_address_data 结构体，这个结构体用用来描述设备地址

```
static struct i2c_client_address_data addr_data = {
	.normal_i2c  = normal_addr,    /*  要发出 S 信号和设备地址并得到 ACK 信号,
才能确定存在这个设备  */
	.probe	         = ignore,
	.ignore	         = ignore,
	//.forces	         = forces, /*  强制认为存在这个设备  */
};
```

3.设备的注册以及探测功能

这一步很关键，按照标准的要求来写，则 linux 系统会自动调用相关的代码去探测你的 i2c 设备，并且添加到系统的 i2c 设备列表中以供后面访问

i2 设备的探测一般是靠设备地址来完成的，那么，首先就需要在驱动代码中声明里要探测的 i2c 设备的地址列表，以及一个宏。

在这里我们自己定义一个数组来代表 i2c 设备的地址(提示:i2c 设备本来的地址是

8 位，但是 linux 和裸机程序是有一定的区别的，这里不能够使用 8 位来代表设备地址，一般用高 7 位来代表 i2c 的设备地址，一般真实的地址右移一位得到 linux 的 i2c 的设备地址)s

```
tatic unsigned short normal_addr[] = { 0x50, I2C_CLIENT_END }; /*  地址值是 7 位 */
```

定义的 i2c 设备的地址数组必须要以 I2C_CLIENT_END 字段进行结尾。

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/jiffies.h>
#include <linux/i2c.h>
#include <linux/mutex.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

static unsigned short ignore[]          = { I2C_CLIENT_END };
static unsigned short normal_addr[] = { 0x50, I2C_CLIENT_END }; /*  地址值是 7 位  */
                                        /*  改为 0x60 的话, 由于不存在设备地址为 0x60 的设备, 所以 at24cxx_detect 不被调用  */


static struct i2c_client_address_data addr_data = {
	.normal_i2c  = normal_addr,   /*  要发出 S 信号和设备地址并得到 ACK 信号, 才能确定存在这个设备  */
	.probe       = ignore,
	.ignore      = ignore,
	//.forces      = forces, /*  强制认为存在这个设备  */
};

static struct i2c_driver at24cxx_driver;


static int major;
static struct class *cls;
struct i2c_client *at24cxx_client;

static ssize_t at24cxx_read(struct file *file, char __user *buf, size_t size, loff_t * offset)
{
```

```
	unsigned char address;
	unsigned char data;
	struct i2c_msg msg[2];
	int ret;

	/* address = buf[0]
	 * data    = buf[1]
	 */
	if (size != 1)
		return -EINVAL;

	copy_from_user(&address, buf, 1);

	/*  数据传输三要素: 源,目的,长度  */

	/*  读 AT24CXX 时,要先把要读的存储空间的地址发给它  */
	msg[0].addr   = at24cxx_client->addr;   /*  目的  */
	msg[0].buf    = &address;               /*  源  */
	msg[0].len    = 1;                      /*  地址=1 byte */
	msg[0].flags = 0;                       /*  表示写  */

	/*  然后启动读操作  */
	msg[1].addr   = at24cxx_client->addr;   /*  源  */
	msg[1].buf    = &data;                  /*  目的  */
	msg[1].len    = 1;                      /*  数据=1 byte */
	msg[1].flags = I2C_M_RD;                /*  表示读  */


	ret = i2c_transfer(at24cxx_client->adapter, msg, 2);
	if (ret == 2)
	{
		copy_to_user(buf, &data, 1);
		return 1;
	}
	else
		return -EIO;
}

static ssize_t at24cxx_write(struct file *file, const char __user *buf, size_t size, loff_t *offset)
{
	unsigned char val[2];
	struct i2c_msg msg[1];
	int ret;
```

```
	/* address = buf[0]
	 * data    = buf[1]
	 */
	if (size != 2)
		return -EINVAL;

	copy_from_user(val, buf, 2);

	/* 数据传输三要素: 源,目的,长度 */
	msg[0].addr  = at24cxx_client->addr;   /* 目的 */
	msg[0].buf   = val;                    /* 源 */
	msg[0].len   = 2;                      /* 地址+数据=2 byte */
	msg[0].flags = 0;                      /* 表示写 */

	ret = i2c_transfer(at24cxx_client->adapter, msg, 1);
	if (ret == 1)
		return 2;
	else
		return -EIO;
}


static struct file_operations at24cxx_fops = {
	.owner = THIS_MODULE,
	.read  = at24cxx_read,
	.write = at24cxx_write,
};

static int at24cxx_detect(struct i2c_adapter *adapter, int address, int kind)
{
	printk("at24cxx_detect\n");

	/* 构构一个 i2c_client 结构体: 以后收改数据时会用到它 */
	at24cxx_client = kzalloc(sizeof(struct i2c_client), GFP_KERNEL);
	at24cxx_client->addr    = address;
	at24cxx_client->adapter = adapter;
	at24cxx_client->driver  = &at24cxx_driver;
	strcpy(at24cxx_client->name, "at24cxx");

	/* 将 i2c_client 于 i2c_adapter 关联起来 */
	i2c_attach_client(at24cxx_client);

	major = register_chrdev(0, "at24cxx", &at24cxx_fops);
```

```c

	cls = class_create(THIS_MODULE, "at24cxx");
	class_device_create(cls, NULL, MKDEV(major, 0), NULL, "at24cxx"); /* /dev/at24cxx */

	return 0;
}

static int at24cxx_attach(struct i2c_adapter *adapter)
{
	/* 初始化适配器，初始化回调函数 */
	return i2c_probe(adapter, &addr_data, at24cxx_detect);
}

static int at24cxx_detach(struct i2c_client *client)
{
	printk("at24cxx_detach\n");
	class_device_destroy(cls, MKDEV(major, 0));
	class_destroy(cls);
	unregister_chrdev(major, "at24cxx");

	i2c_detach_client(client);
	kfree(i2c_get_clientdata(client));

	return 0;
}


/* 1. 分配一个 i2c_driver 结构体 */
/* 2. 设置 i2c_driver 结构体 */
static struct i2c_driver at24cxx_driver = {
	.driver = {
		.name	= "at24cxx",
	},
	.attach_adapter = at24cxx_attach,  /* 这个函数是相当于 probel 的作用 */
	.detach_client	= at24cxx_detach,
};

static int at24cxx_init(void)
{
	i2c_add_driver(&at24cxx_driver);
	return 0;
}

```

```
static void at24cxx_exit(void)
{
	i2c_del_driver(&at24cxx_driver);
}

module_init(at24cxx_init);
module_exit(at24cxx_exit);

MODULE_LICENSE("GPL");
```

Spi 驱动编写

spi 驱动程序的编写(参考内核代码)

在 linux 内核中，spi 有系统自带的设备驱动程序-spidev.c。在编写 spi 驱动程序的时候我们也要按照板级-总线-驱动的方式进行编写。内核已经包含了 spi 控制器驱动，我们只需要编写设备驱动程序和初始化 spi_master

这一点和 i2c 是一样的，spi 驱动程序的编写和 i2c 采用的是一样的方式编写。

spi 驱动由三部分组成，分别是 core，spi_master 以及 spi_drivers(一个 spi_master 代表一个 spi 主控制器，这部分驱动一般不用我们自己编写，linux 内核一般自带了。)

一个 spi_device 代表一个外围的 spi 的设备，spi_master 注册完成之后会扫描 bsp 中注册的 spi 设备链表，并向 spi_bus 注册。

spi 的读写是调用 spi_transfer 来完成的，它通过构造 spi message 来实现的，这一点类似于 i2c 设备的发送与接收。

Spi_message 描述一次完整的传输,即 cs 信号线从高-低-高这样一个过程。

spi_message 代表了 spi 的下消息，它由多个 spi_transfer 段组成的，这个传输队列是原子的，意思就是这个消息在传递完成之前不会被抢占。Spi_message 由多个 spi_transfer 构成，例如我们对于 spi 外设的读操作可以分解成为两个 spi_transfer，一个是写命令，一个是读数据。

1.在板级文件中定义具体的 spi 设备所需要的硬件信息(填充 spi_board_info 结构用于初始化 spi_device)

```
static struct ads7846_platform_data ads_info = {
	.model		= 7843,
	.x_min		= 150,
	.x_max		= 3830,
	.y_min		= 190,
	.y_max		= 3830,
	.vref_delay_usecs	= 100,
	.x_plate_ohms		= 450,
	.y_plate_ohms		= 250,
	.pressure_max		= 15000,
	.debounce_max		= 1,
	.debounce_rep		= 0,
	.debounce_tol		= (~0),
	.get_pendown_state	= ads7843_pendown_state,
};
static struct spi_board_info ek_spi_devices[] = {
#if defined(CONFIG_MTD_AT91_DATAFLASH_CARD)
	{	/* DataFlash card */
```

```
		.modalias	= "mtd_dataflash",
		.chip_select	= 0,
		.max_speed_hz	= 15 * 1000 * 1000,
		.bus_num	= 0,
	},
#endif
#if defined(CONFIG_TOUCHSCREEN_ADS7846) || defined(CONFIG_TOUCHSCREEN_ADS7846_MODULE)
	{
		.modalias	= "ads7846",
		.chip_select	= 3,
		.max_speed_hz	= 125000 * 26,	/* (max sample rate @ 3V) * (cmd + data + overhead) */
		.bus_num	= 0,
		.platform_data	= &ads_info,
		.irq		= AT91SAM9263_ID_IRQ1,
	},
#endif
};
```

说明 spi_board_info 根据 linux 的内核版本不同定义也有所不同,但是大体上是相同的

```
struct spi_board_info {
	char		modalias[32];
	const void	*platform_data;
	void		*controller_data;
	int		irq;
	u32		max_speed_hz;
	u16		bus_num;
	u16		chip_select;
	u8		mode;
};
```

说明：第一个参数设备的名称，第二个参数是用来传输的私有的数据，第三个参数可以用来作为 cs 引脚的控制函数的入口地址，第四个参数是中断，第五个参数是最大的传输频率，第六个参数是 bus 号(实际指的是对应的 spi 的寄存器)，第七个参数是片选信号(这里的片选信号是片选引脚编号的索引，编号索引不超过 num_cs 的数目，那么真正的片选引脚在什么地方声明的呢？)，第八个参数是 spi 的传输模式。

2.编写驱动程序(主要参考内核代码的 spidev.c-这里需要感谢 2 群的一位同学，我也参考了他发给我的 spi 的驱动代码)

构造 spi_srivers 数据结构

```
static struct spi_driver spidev_spi = {
	.driver = {
```

```
		.name =		"spidev",
		.owner =	THIS_MODULE,
	},
	.probe = spidev_probe,
	.remove =	__devexit_p(spidev_remove),

	/* NOTE:  suspend/resume methods are not necessary here.
	 * We don't do anything except pass the requests to/from
	 * the underlying controller.  The refrigerator handles
	 * most issues; the controller driver handles the rest.
	 */
};
```

说明:这里的参数我不再说明。

注册 spi_driver 结构体

```
status = spi_register_driver(&spidev_spi);
```

说明:在 linux 中你必须遵守一个规则，定义了一般就需要注册。

定义一个操作函数集，然后注册进内核

```
static struct file_operations spidev_fops = {
	.owner =	THIS_MODULE,
	/* REVISIT switch to aio primitives, so that userspace
	 * gets more complete API coverage.  It'll simplify things
	 * too, except for the locking.
	 */
	.write =	spidev_write,
	.read =		spidev_read,
	.unlocked_ioctl = spidev_ioctl,
	.open =		spidev_open,
	.release =	spidev_release,
};
status = register_chrdev(SPIDEV_MAJOR, "spi", &spidev_fops);
/* 注册操作函数集 */
```

3.一旦设备文件和驱动匹配，那么就会调用 probel 函数

在设备函数中我们最重要的几个步骤就是得到设备文件描述的设备信息，创建一个字符设备

```
static int spidev_probe(struct spi_device *spi)
{
	struct spidev_data	*spidev;
	int			status;
	unsigned long		minor;

```

```
	/* Allocate driver data */
	spidev = kzalloc(sizeof(*spidev), GFP_KERNEL);
	if (!spidev)
		return -ENOMEM;

	/* Initialize the driver data */
	spidev->spi = spi;
	spin_lock_init(&spidev->spi_lock);
	mutex_init(&spidev->buf_lock);
	INIT_LIST_HEAD(&spidev->device_entry);

	/* If we can allocate a minor number, hook up this device.
	 * Reusing minors is fine so long as udev or mdev is working.
	 */
	mutex_lock(&device_list_lock);
	minor = find_first_zero_bit(minors, N_SPI_MINORS);
	if (minor < N_SPI_MINORS) {
		struct device *dev;

		spidev->devt = MKDEV(SPIDEV_MAJOR, minor);
		dev = device_create(spidev_class, &spi->dev, spidev->devt,
		  spidev, "spidev%d.%d",
		  spi->master->bus_num, spi->chip_select);
		status = IS_ERR(dev) ? PTR_ERR(dev) : 0;
	} else {
		dev_dbg(&spi->dev, "no minor number available!\n");
		status = -ENODEV;
	}
	if (status == 0) {
		set_bit(minor, minors);
		list_add(&spidev->device_entry, &device_list);
	}
	mutex_unlock(&device_list_lock);

	if (status == 0)
		spi_set_drvdata(spi, spidev);
	else
		kfree(spidev);
	return status;
}
```

说明；上面的主要的步骤已经有红色字体标注，其他的步骤并不是最重要的，可以结合内核的 spidev.c 驱动程序里面看。

4.编写 open，read，write 函数

Open 函数一般是创建字符设备，这里不再讲解。

重点分析的是 write 函数以及 read 函数，下面一一进行讲解(摘录自内核的—可以参考 spidev.c 文件)

```
static int m25p80_read(struct mtd_info *mtd, loff_t from, size_t len,
	size_t *retlen, u_char *buf)
{
	struct m25p *flash = mtd_to_m25p(mtd);
	struct spi_transfer t[2];
	struct spi_message m;
	DEBUG(MTD_DEBUG_LEVEL2, "%s: %s %s 0x%08x, len %zd\n",
			dev_name(&flash->spi->dev), __func__, "from",
			(u32)from, len);
	/* sanity checks */
	if (!len)
		return 0;
	if (from + len > flash->mtd.size)
		return -EINVAL;
	spi_message_init(&m);   /*  初始化 spi_message */
	memset(t, 0, (sizeof t));
	/* NOTE:
	 * OPCODE_FAST_READ (if available) is faster.
	 * Should add 1 byte DUMMY_BYTE.
	 */
	t[0].tx_buf = flash->command;     /*  命令  */
	t[0].len = CMD_SIZE + FAST_READ_DUMMY_BYTE;    /*  定义第一个
transfer 的写指针和长度  */
	spi_message_add_tail(&t[0], &m);          /*  添加到 spi_message 中  */
	t[1].rx_buf = buf;      /*  读的数据保存的缓冲区  */
	t[1].len = len;     /*  第二个 transfer 的读指针以及长度  */
	spi_message_add_tail(&t[1], &m);          /*  添加到 spi_message 中  */

	/* Byte count starts at zero. */
	if (retlen)
		*retlen = 0;
	mutex_lock(&flash->lock);
	/* Wait till previous write/erase is done. */
	if (wait_till_ready(flash)) {
		/* REVISIT status return?? */
		mutex_unlock(&flash->lock);
		return 1;
	}
	/* FIXME switch to OPCODE_FAST_READ.   It's required for higher
```

```
         * clocks; and at this writing, every chip this driver handles
         * supports that opcode.
         */
        /* Set up the write data buffer. */
        flash->command[0] = OPCODE_READ;
        flash->command[1] = from >> 16;
        flash->command[2] = from >> 8;
        flash->command[3] = from;
        spi_sync(flash->spi, &m);      /*  调用 spi_masster 发送 spi_message */
        *retlen = m.actual_length - CMD_SIZE - FAST_READ_DUMMY_BYTE;
        mutex_unlock(&flash->lock);
        return 0;
}
```

说明:在编写 read 函数的时候，主要涉及到一下几个步骤:

1．定义 spi_message
2．定义 spi_transfer
3．初始化 spi_message
4．填充 spi_transfer（对于读的操作需要两个步骤，1.读取的 spi 外设的寄存器的地址，2，读取的值保存的 buf。）
5．启动 spi 发送 spi_sync:->spi_sync 为同步发送，在调用这个函数的时候就开始传输，还可以用 spi_async 异步方式，就是调了这个函数但是这个函数的执行时间不确定，所以在使用异步方式的时候需要指定回调函数。我们在这里也可以选择一些封装的更好的函数进行调用 int spi_write_then_read(struct spi_device *spi,const u8 *txbuf, unsigned n_tx,u8 *rxbuf, unsigned n_rx)。第一个参数是 spi 设备地址，第二个参数是需要发送的数据，第三个参数是发送数据的长度，第四个参数是接收的数组，第五个参数是需要接收数据的字节数。(关于这个函数怎么使用将在后面进行说明)。

```
static int m25p80_write(struct mtd_info *mtd, loff_t to, size_t len,
        size_t *retlen, const u_char *buf)
{
        struct m25p *flash = mtd_to_m25p(mtd);
        u32 page_offset, page_size;
        struct spi_transfer t[2];
        struct spi_message m;
        DEBUG(MTD_DEBUG_LEVEL2, "%s: %s %s 0x%08x, len %zd\n",
                        dev_name(&flash->spi->dev), __func__, "to",
                        (u32)to, len);
        if (retlen)
                *retlen = 0;
        /* sanity checks */
        if (!len)
                return(0);
```

```
	if (to + len > flash->mtd.size)
		return -EINVAL;
	spi_message_init(&m);
	memset(t, 0, (sizeof t));
	t[0].tx_buf = flash->command;
	t[0].len = CMD_SIZE;
	spi_message_add_tail(&t[0], &m);
	t[1].tx_buf = buf;
	spi_message_add_tail(&t[1], &m);
	mutex_lock(&flash->lock);
	/* Wait until finished previous write command. */
	if (wait_till_ready(flash)) {
		mutex_unlock(&flash->lock);
		return 1;
	}
	write_enable(flash);
	/* Set up the opcode in the write buffer. */
	flash->command[0] = OPCODE_PP;
	flash->command[1] = to >> 16;
	flash->command[2] = to >> 8;
	flash->command[3] = to;
	/* what page do we start with? */
	page_offset = to % FLASH_PAGESIZE;
	/* do all the bytes fit onto one page? */
	if (page_offset + len <= FLASH_PAGESIZE) {
		t[1].len = len;
		spi_sync(flash->spi, &m);
		*retlen = m.actual_length - CMD_SIZE;
	} else {
		u32 i;
		/* the size of data remaining on the first page */
		page_size = FLASH_PAGESIZE - page_offset;
		t[1].len = page_size;
		spi_sync(flash->spi, &m);
		*retlen = m.actual_length - CMD_SIZE;
		/* write everything in PAGESIZE chunks */
		for (i = page_size; i < len; i += page_size) {
			page_size = len - i;
			if (page_size > FLASH_PAGESIZE)
				page_size = FLASH_PAGESIZE;

			/* write the next page to flash */
			flash->command[1] = (to + i) >> 16;
			flash->command[2] = (to + i) >> 8;
```

```
				flash->command[3] = (to + i);
				t[1].tx_buf = buf + i;
				t[1].len = page_size;
				wait_till_ready(flash);
				write_enable(flash);
				spi_sync(flash->spi, &m);
				if (retlen)
					*retlen += m.actual_length - CMD_SIZE;
			}
		}
		mutex_unlock(&flash->lock);
		return 0;
	}
```

接收的过程和发送的过程类似，这里不再重复说明

驱动程序的编写大概就是这些，下面分析用户态的应用程序

5.spi 的应用程序和 i2c 也极其的类似，都是在用户空间构造 message 结构体，在 linux 内核中已经有了 spi 的用户程序，程序位于内核目录的(Documentation/spi/spitest.c)，用户态的程序不再进行分析。(用户态主要填充 spi_ioc_transfer 结构体)。

说明:在我们编写 spi 的驱动程序的时候，完全可以修改 spidev.c 中的代码，在 linux 内核中一个 spi 的主控制器最多能够挂载 32 个 spi 的外设，所有的 spi 的外设的驱动程序都可以按照 spidev.c 的内核驱动修改得到，我们只需要填写对应的板级信息，但是 cs 引脚应该怎么指定？在 spi_board_info 结构体中没有指定 cs 的引脚，只有一个 select_cs 这个引脚是 cs 引脚的索引编号，不代表真正的 spi 设备的 cs 引脚。一个 spi 控制器上挂载的 spi 外设的 select_cs 一定不能一样，你可以从 0 依次递增，但是绝对不能够超过该控制器上所能够支持的外设总数(一般是在 spi_master 结构体中设置)，根据 arm 芯片的设计，一个 spi 控制器只能挂载 32 个 spi 的外设。

DMA 驱动程序的编写(参考韦东山的驱动代码)

Dma 一般用在大容量数据的传输模式中，比如说视频流等，在 cpu 使能 dma 模式之后，能够不需要 cpu 的参与就能够在内存-内存，内存-外设，外设-内存，外设到外设之间传递数据，dma 的工作模式如下:
I/O 向 DMAC 发送请求->DMAC 向 CPU 发送请求->CPU 相应 DMAC 的请求->DMAC 向 IO 发送请求->DMAC 发送内存地址->DMAC 发出控制信号->DMA 开始传输->DMAC 传输结束。
注意:DMA 是在两个物理地址间传输数据。

在 linux 内核中，如果要编写 dma 的驱动程序，一般需要调用一下的一系列初始化函数:
Dma_cap_zero，dma_cap_set，dma_request_channel，申请 buf，sg_init_one，device_control，dma_map_sg，device_prep_slave_sg，dmaengine_submit，device_issue_pending 等函数。

在我们自己写的驱动程序中可以按照一下等方式来编写我们的 dma 驱动程序。
request_irq(IRQ_DMA3, s3c_dma_irq, 0, "s3c_dma", 1)   /* 申请 dma 中断 */
说明：在我们申请中断的时候，寄存器中断与外部中断是不同的，寄存器中断的触发条件直接填 0，外部中断有触发条件，我们可以选择低电平触发，高电平触发，下降沿触发，上升沿触发，双边沿触发等，寄存器中断的编写和外部中断的编写时截然不同的，在实际的使用中需要注意。

2．申请内存空间:dma_alloc_writecombine
在这里我们假设是在内存-内存之间传递数据。那么我们需要申请 src 内存空间和 dst 内存空间(寄存器到内存，寄存器到寄存器之间的程序讲解会在后面进行说明。)